REMARKS

I. Introduction

In response to the Office Action dated February 8, 2005, which was made final, claims 1, 6, 8, 12, 17, 19, 23, 28, 30 and 34 have been amended, and new claims 41-50 have been added. Claims 1-4, 6, 8-10, 12-15, 17, 19-21, 23-26, 28, 30-32, 34-39 and 41-50 are in the application. Entry of these amendments, and re-consideration of the application, as amended, is requested.

II. Interview Summary Record

Record is made of telephonic interviews that occurred on March 23, 2005 and April 7, 2005 between Examiner Nalven and the below-signed attorney. Possible claim amendments were discussed, along with the prior art references, but no agreement was reached.

III. Prior Art Rejections

A. The Office Action Rejections

In paragraphs (3)-(4) of the Office Action, claims 1-3, 6, 8-10, 12-14, 17, 19-21, 23-25, 28, 30-32, 34, and 36-38 were rejected under 35 U.S.C. §103(a) as being unpatentable over Hunt, U.S. Patent No. 6,154,747 (Hunt) in view of Fischer, U.S. Patent No. 6,105,072 (Fischer) and further in view of Morel et al., U.S. Patent No. 5,721,919 (Morel). In paragraph (7) of the Office Action, claims 4, 15, 26, 35, and 39 were rejected under 35 U.S.C. §103(a) as being unpatentable over Hunt, Fischer, and Morel as applied to claims 1, 23, and 34, and further in view of Moore, U.S. Patent No. 5,343,527 (Moore).

Applicants' attorney respectfully traverses these rejections.

B. The Applicants' Independent Claims

Independent claims 1, 12 and 23 are generally directed to checking a version of an abstract data type stored in a database. Independent claim 1 is representative, and comprises:

(a) constructing an identifier for the abstract data type, wherein the identifier comprises a concatenation of information describing the abstract data type that is substantially unique to the abstract data type;

(b) hashing the constructed identifier to generate a signature hash value for the abstract data type;

(c) storing the signature hash value in the database and a class definition for the abstract data type; and

(d) comparing the signature hash value from the database with the signature hash value from the class definition after the class definition is instantiated in order to verify that the class definition is not outdated.

Independent claims 6, 17 and 28 are generally directed to generating a signature hash value for checking a version of an abstract data type stored in databases. Independent claim 6 is representative, and comprises:

(a) constructing an identifier for the abstract data type, wherein the identifier comprises a concatenation of information describing the abstract data type that is substantially unique to the abstract data type;

(b) hashing the constructed identifier to generate a signature hash value for the abstract data type; and

(c) storing the signature hash value in the database and a class for the abstract data type.

Independent claims 8, 19 and 30 are generally directed to verifying a signature hash value for checking a version of an abstract data type stored in a database. Independent claim 8 is representative, and comprises:

(a) accessing a first signature hash value from the database and a second signature hash value from a class definition for the abstract data type, wherein the first and second signature hash values have been constructed from an identifier for the abstract data type, the identifier comprises a concatenation of information describing the abstract data type that is substantially unique to the abstract data type, and the identifier has been hashed to generate the first and second signature hash values; and

(b) comparing the first signature hash value with the second signature hash value after the class definition is instantiated in order to verify that the class definition is not outdated.

Independent claim 34 is directed to at least one data structure stored in a memory for use in checking a version of an abstract data type stored in a database, the data structure comprising a signature hash value for the abstract data type generated from an identifier constructed for the abstract data type, wherein the identifier comprises a concatenation of information describing the abstract data type that is substantially unique to the abstract data type and the identifier is hashed to generate the signature hash value for the abstract data type.

C. The Hunt Reference

Hunt describes a method using a plurality of hash tables to provide an object repository for object oriented application development and use. The method includes storing an object identifier and a representation of the object in a first hash table and storing data about the object and the object identifier in a plurality of paired hash tables with the hash tables organized in mirrored table pairs. The data about the object include an object's class name, object methods, return types, and the data values returned by object methods. The non-inverse hash tables of the mirrored table pairs support fuzzy searches for objects while the inverse hash tables of the mirrored table pairs support searches for objects by object identifier. A system that implements the inventive method includes a first hash table for storage of data representing the object with an object identifier used as a key for storage of the representing data and a plurality of mirrored table pairs for the storage of data about objects. In the non-inverse table of the mirrored table pairs, the data about the objects are used as keys to store an object identifier. In the inverse tables of the mirrored table pairs, the object identifier is used as the key for storage of the data about the data objects. The mirrored table pairs and hash table provide an efficient and economical object repository. Such an object repository may be provided with a computer application at a nominal expense. Thus, object oriented applications may be developed without regard for whether the end user system includes an object repository.

D. The Fischer Reference

Fischer describes a method of operating computers in accordance with an enhanced object-oriented programming methodology that creates a framework for efficiently performing automated business transactions. The object-oriented programming methodology is used in conjunction with a travelling program, i.e., a digital data structure which includes a sequence of instructions and associated data which has the capability of determining at least one next destination or recipient for receiving the travelling program and for transmitting itself, together with all relevant data determined by the program to the next recipient or destination Using the methods described herein, the data is more closely bound to the program in such a way that objects may be most efficiently transferred from one computer user to another without the objects being previously known to the recipient computer user. The present invention utilizes object "cells" which are data structures stored, for example, on a disk that reflects a collection of (related) objects instances whose execution has been suspended, and which can be resumed later on the same or a different platform. The collection of object instances can be gathered together into cells (or "electronic forms") suitable for storage or transmission to another computer user in such a way that instances are unambiguously bound to their respective class definition. The present invention also creates improved tools for creating and using cells so that electronic forms can be defined using object-oriented techniques while allowing such forms to be easily transferred among a diverse population of computer users--without demanding that all users maintain compatible libraries of all object-class definition programs and without demanding that all users maintain identical synchronized versions of that class. The invention provides a digital signature methodology to insure security and integrity, so that electronic forms (i.e., cells) composed of a collection of objects can be received and executed by a user without putting the user at risk that some of the object classes embedded in the cell might be subversive "trojan horse" programs that might steal, destroy or otherwise compromise the security or integrity of the user's system or data.

E. The Morel Reference

Morel describes a method and system for tracking, and resolving links to, objects that derive from a common object creation. In a system, the system creates a source object. The system then generates a lineage identifier to identify the creation of the source object. Then the system associates the lineage identifier with the source object. At a later time, the system copies the created object to a copy object. When the source object is copied to a copy object, the system associates the lineage identifier associated with the source object with the copy object. In this way, the lineage identifier associated with the copy object indicates that the copy object derives from the creation of the source object. The system links a client object to a source object by storing a link containing the source object's lineage identifier in the client object. A link also contains information for distinguishing the source object from other objects having the same lineage identifier. When resolving the link to the source object, the system selects the lineage identifier and the distinguishing information contained in the link. The system then searches for an object with the selected lineage identifier and distinguishing information. When an object with the selected lineage identifier and distinguishing information is found, the system resolves the link to the found object. When an object with the selected lineage identifier and distinguishing information is not found, the system searches for an object with the selected lineage identifier without regard to the selected distinguishing information. When an object with the selected lineage identifier is found, the system resolves the link to this found object.

F. The Moore Reference

Moore describes a system and method for providing a reuser of a software reuse library with an indication of whether or not a software component from the reuse library is authentic and whether or not the software component has been modified. The system and method disclosed provides a reuser with assurance that the software component retrieved was placed in the reuse library by the original publisher and has not modified by a third party. The system and method disclosed uses a hybrid cryptographic technique that combines a conventional or private key algorithm with a public key algorithm.

G. The Applicants' Claims Are Patentable Over The References

Applicants' invention, as recited in the claims, is patentable over the references, because the claims recite limitations not found in the references.

On the other hand, the Office Action asserts that Hunt teaches the construction of an identifier for the abstract data type where the identifier is substantially unique to the data type and the identifier comprises a concatenation of various attributes for the data type, at column 6, lines 43-45, column 6, lines 45-50, column 7, lines 1-3 and column 7, lines 42-49.

However, at the cited locations, Hunt merely discloses the following:

> Hunt, column 6, lines 43-50 (actually, lines 29-67)
>
> A preferred structure for object repository 18 that may be accessed by an instance of the gateway depicted in FIG. 3 is shown in FIG. 4. Object repository 18 includes a single hash table 62 and hash table pairs 64, 68, 72, and 74 which are organized in mirrored table pairs. Each mirrored table pair is comprised of a non-inverse hash table and an inverse hash table. The inverse tables are so referenced because they reverse the key/value organization of the non-inverse hash tables. Mirrored table pair 64 stores class names for objects. Mirrored table pair 68 stores data regarding the methods used by the objects stored in repository 18 and mirrored table pair 72 stores data regarding return types for the methods of the objects stored in repository 18. Mirrored table pair 74 is used to store data values returned by the methods for objects stored in repository 18. Hash table 62 is organized to store object data using an object identifier as a key. Preferably, the database used to implement the hash tables of repository 18 generate the key for storing information in a hash table by providing an object identifier (OID) as an argument to a hashing function. The value generated by the hashing function is used as a key for storing data in a hash table. Because use of serialized data object data would not be efficiently hashed to generate a key, the hash table used to store object data does not have a corresponding inverse table. Instead, an OID is used as a key to store object data in hash table 62 after the object data has been serialized, that is, converted to string data. An object's class name is stored in the non-inverse table 64a as the key

for the OID of the corresponding data object. Likewise, object methods are stored in non-inverse table 68a as the key for the OID of the corresponding object and the return types for the methods of the objects are stored in the non-inverse table 72a as the key for the OID of the corresponding object. The data value returned by a method of an object is concatenated with the object's class name and method to generate a key that is used to store the OID in non-inverse table 74a. In the inverse table of each of the mirrored table pairs, the OID is used as the key to store the data value that is the key for the OID in the non-inverse table.

Hunt, column 7, lines 1-3 (actually, lines 1-28)

Preferably, each hash table of object repository 18 is implemented with a Berkeley DB database system available from Sleepycat Software Inc. of Carlisle, Mass. and the programming language statements that implement instances of interface 20 communicate with the management system for the Berkeley DB through a thread extending to repository 18. The Berkeley DB database uses union data structures implemented in the C programming language. Preferably, one Berkeley DB is used for each of the nine hash tables required to implement the preferred structure discussed above. Eight of the Berkeley DBs are used for the four mirrored table pairs and the other Berkeley DB is used for the single hash table in the preferred implementation. Berkeley DBs may be configured for one of three access methods. These access methods are binary tree (mode 1), extended linear hash table (mode 2), and fixed or variable length record (mode 3). Preferably, the Berkeley databases are configured to operate in mode 2 for use as hash tables. The programming statements that implement interface 20 preferably include five (5) instances of a Java class with four of the five class instrances controlling access to a mirrored table pair and the fifth instance of the class controlling access to the single hash table. The Berkeley DB preferably used to implement repository 18 also includes a management system that handles transactions, locking, logging, shared memory caching and database recovery. In addition, Berkeley DB supports C, C++, Java and Perl application programming interfaces.

Hunt, column 7, lines 42-49 (actually, lines 42-65)

Object identifiers are preferably generated by capturing system time in milliseconds since a certain date in 1970 and concatenating that value with the number of objects processed by the instance of interface 20 being used by an application 14. This object identifier (OID) is preferably generated in string form and is returned to application programs 14 as well as being used to store data in object repository 18. However, in distributed environments, two instances of interface 20 may have processed the same number of data objects and may generate an object identifier at the same time. Accordingly, both instances would generate the same object identifier. As a result, the object identifier may be provided to the wrong instance of interface 20 for retrieval of the object without detection or an attempt to store different objects in the same repository with the same OID may occur. To reduce the likelihood of this occurrence, generation of the OID is performed as discussed above except a prefix string corresponding to the name of the server on which an instance of interface 20 is executing is concatenated to the OID. As instances of interface 20 likely execute on different servers in a distributed environment, this method of OID generation addresses the remote likelihood of the same OID being generated by different instances of interface 20.

The above portions of Hunt merely state that the object identifier (OID) is generated by capturing system time in milliseconds since a certain date in 1970, concatenating that value with the number of objects processed by the instance of interface 20 being used by an application 14, and prefixing a string corresponding to the name of the server on which an instance of interface 20 is executing. However, Hunt does not teach or suggest constructing an identifier from a concatenation of information describing an abstract data type that is substantially unique to the abstract data type. Specifically, the system time, number of objects processed, and name of the server do not comprise "information describing the abstract data type."

The Office Action also asserts that Morel teaches the storing of the hash value in the class definition, at column 8, lines 57-63 and column 5, lines 33-62.

However, at the cited locations, Morel merely discloses the following:

> <u>Morel, column 8, lines 57-63 (actually, column 8, line 51 – column 9, line 2)</u>
> When the facility generates an object identifier (including a lineage identifier and a distinguished identifier) for an object, it associates that object identifier with the object so that (a) when a user decides to establish a link to the object, the facility can establish a link containing the object identifier; and (b) the facility can search for objects having a certain object identifier when resolving the link. In a preferred embodiment, when the facility associates an object identifier with an object, it stores the object identifier inside the object. In this way, the object knows its own object identifier. The facility preferably establishes an object identifier table that maps object identifiers to the pathnames of the associated objects. When an object identifier is generated for an object, the facility updates the object identifier table to include a mapping of that object identifier to the pathname of the object. If the user moves or renames the object, the facility updates the pathname stored in the object identifier table. Storing the object identifiers in a table permits the facility to quickly search for objects.
>
> <u>Morel, column 5, lines 33-62</u>
> The system links a client object to a source object by storing a link containing the source object's lineage identifier in the client object. A link also contains information for distinguishing the source object from other objects having the same lineage identifier. When resolving the link to the source object, the system selects the lineage identifier and the distinguishing information contained in the link. The system then searches for an object with the selected lineage identifier and distinguishing information. When an object with the selected lineage identifier and distinguishing information is found, the system resolves the link to the found object. When an object with the selected lineage identifier and distinguishing information is not found, the system searches for an object with the selected lineage identifier

without regard to the selected distinguishing information. When an object with the selected lineage identifier is found, the link system resolves the link to this found object.

When resolving a link, the system preferably searches for the source object of the link in a series of volumes in an optimal order. The system preferably first checks a pathname stored in the link, then searches a hinted volume, then searches all local volumes, then searches volumes in an automatic volume list, then searches volumes in a manual volume list, then searches volumes in remote volume lists indicated by a list of remote volume lists, then broadcasts a search request to all connected machines. The system preferably also implements an object identifier table that maps from object identifiers used in links directly to file system identifiers, thereby bypassing the step of looking up a source object file name in a directory specified by a pathname.

The above portions of Morel merely describe a lineage identifier for a source object being associated with a copy object to indicate that the copy object derives from the source object. However, Morel does not teach or suggest storing a signature hash value both in a database and a class definition for the abstract data type, so that at a later time the signature hash value from the database can be compared with the signature hash value from the class definition, after the class definition is instantiated, in order to verify that the class definition is not outdated.

Finally, the Office Action asserts that Fischer teaches the comparing of signature hash values from the database with signature hash values from the class definition, at column 30, lines 24-44 and 55-58, column 30, line 66 – column 31, line 12, and column 4, lines 19-49.

However, at the cited locations, Fischer merely discloses the following:

<u>Fischer, column 30, lines 24-44 (actually, lines 55-65)</u>

In reloading a class, it is necessary to reestablish the class exactly as it existed before. The reload class routine begins at block 1202 where a new classBlock is created which corresponds to the prospective load (block will be deleted in case of failure). the classBlock is connected to the class list and an indication is made as to whether the classBlock is incompletely loaded as previously described to detect circular inheritance.

A check is made at block 1206 to determine whether class p-code definition is already provided in the cell. If so, then the routine branches to block 1230 and processes the p-code image as carried in the cell. If the p-code definition is not present, then a check is made to determine if the class source code definition is provided in the cell. If so, then the routine branches to 1220 and processes the source image as carried in the cell. If neither the p-code not source code is present, then the routine proceeds to block 1210 where the class definition is indicated by program name, by program version, by p-code and source hash. The name (and possibly the hash) of the class is used to locate (another) local copy of the p-code or the source.

Fischer, column 30, lines 55-58 (actually, lines 55-65)

If the source code has been located, then at block 1220, it is compiled and the hash of the source is computed. If the computed source hash matches that defined in the authenticated aspect of the cell, then the class is accepted. It is possible that the source hash is based not on the exact source, but rather a normalized form, for example, with the comments removed and nonessential spacing deleted. Thereafter, in block 1220, the class authorization is established associated with the source code. The source definition is compiled using whatever compiler is appropriate to the source definition if multiple languages are supported.

Fischer, column 30, line 66 – column 31, line 12

Processing then continues in block 1232. If the prior processing involved a block 1230, then the located version of the p-code is loaded from the local file repository or within the cell itself. The hash of the p-code is computed as it is loaded. A check is made to ensure that the precompiled p-code is in a format eligible for being processed by this interpreter/executor and the class authorization is established that is associated with the p-code. A check is made to determine whether the computed hash of the p-code equals the expected value as defined in the authenticated part of the cell. If there is a match, then the class is accepted. If not, an error indicator is generated. The processing at 1232 additionally saves the hash of the original source which is given in p-code.

Fischer, column 4, lines 19-49

This provides integrity by insuring that changed "library" or template versions of a class program cannot inadvertently operate on older instance data; or that other incorrect versions of a class program cannot be inadvertently used (and cause confusion or damage) when a cell is activated at different times by a variety of users (recipients) [even if one of the users may have a class program with a matching name]. The unambiguous binding between instance and class may be provided in a variety of ways including:

by binding an object instance in a cell to its class by including the class definition--i.e., the class program logic itself, whether it be in source, p-code or matching code--as part of the cell data structure.

By binding an object instance in a cell to its class by including in the cell a [cryptographic] HASH of at least one of: the SOURCE instructions (or normalized version thereof); the pseudo-code (p-code) instructions resulting from compilation; or the machine language code resulting from compilation--for the class program definition.

The binding correlates to each instance with precisely the correct corresponding class program--so that another class with the same name, or an anachronistic version (too old or too new) of the class--cannot be inadvertently selected to operate on the existing version of instance data, when the cell is reloaded. This is especially useful for class programs that perform critical or sensitive functions. In this fashion, "master" class definitions may be changed without impairing or confusing existing instances that depend on the specification of the class definition at the time the instance was created.

The above portions of Fischer merely describes how the source code of a class definition is hashed, but this hashing function is performed each time the source code is located and is not stored in the source code. However, Fischer does not teach or suggest storing a signature hash value both in a database and a class definition for the abstract data type, so that at a later time the signature hash value from the database can be compared with the signature hash value from the class definition, after the class definition is instantiated, in order to verify that the class definition is not outdated.

Consequently, even when combined, Hunt, Fischer and Morel do not teach or suggest the limitations of Applicants' claimed invention. Moreover, the various elements of Applicants' claimed invention together provide operational advantages over the references. In addition, Applicants' invention solves problems not recognized by the references.

Thus, Applicants' attorney submits that independent claims 1, 6, 8, 12, 17, 19, 23, 28, 30, and 34 are allowable over Hunt, Fischer and Morel. Further, dependent claims 2-4, 9-10, 13-15, 20-21, 24-26, 31-32, 35-39 and 41-50 are submitted to be allowable over the references in the same manner, because they are dependent on independent claims 1, 6, 8, 12, 17, 19, 23, 28, 30, and 34, respectively, and thus contain all the limitations of the independent claims. In addition, dependent claims 2-4, 9-10, 13-15, 20-21, 24-26, 31-32, 35-39 and 41-50 recite additional novel elements not shown by the references.

IV. Conclusion

In view of the above, it is submitted that this application is now in good order for allowance and such allowance is respectfully solicited.

Should the Examiner believe minor matters still remain that can be resolved in a telephone interview, the Examiner is urged to call Applicants' undersigned attorney.

Respectfully submitted,

GATES & COOPER LLP
Attorneys for Applicants

Howard Hughes Center
6701 Center Drive West, Suite 1050
Los Angeles, California 90045
(310) 641-8797

Date: April 8, 2005

By: [illegible]
Name: George H. Gates
Reg. No.: 33,500

GHG/

G&C 30571.231-US-U1